Probabilistic Noninterference in a Concurrent Language [†]

Dennis Volpano
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943, USA
`volpano@cs.nps.navy.mil`

Geoffrey Smith
School of Computer Science
Florida International University
Miami, FL 33199, USA
`smithg@cs.fiu.edu`

Abstract

In [15]*, we give a type system that guarantees that well-typed multi-threaded programs are possibilistically noninterfering. If thread scheduling is probabilistic, however, then well-typed programs may have probabilistic timing channels. We describe how they can be eliminated without making the type system more restrictive. We show that well-typed concurrent programs are probabilistically noninterfering if every total command with a high guard executes atomically. The proof uses the concept of a probabilistic state of a computation, following the work of Kozen* [10].[1]

1. Introduction

This work is motivated by applications of mobile code where programs are downloaded, as needed, and executed on a trusted host (examples include web browsers and e-commerce applications for smartcards and set-top boxes). Here a host may have sensitive data that downloaded code may need, and we want assurance that they are not leaked by the code. In some cases, the best approach may simply be to forbid any access to the sensitive data, using some access control mechanism. But often the code will legitimately need to access the data in order to function. In this case, we need to ensure that it is not leaked by the code.

Specifically, this paper is concerned with identifying conditions under which concurrent programs can be proved free of probabilistic timing channels. Previous work has centered around developing a type system for which one can prove that well-typed multi-threaded programs have a possibilistic noninterference property [15]. The proof relies on a purely nondeterministic thread-scheduling semantics. But although the property rules out *certainty* in deducing private data, its practical utility is somewhat questionable. The trouble is that thread scheduling is usually probabilistic in real implementations, and in this case it is easy to construct well-typed programs with probabilistic timing channels. Here we show how to rule out such channels without making the type system more restrictive.

1.1. The basic idea

Consider a simple imperative language with threads where each thread is a sequence of commands and threads are scheduled nondeterministically. A thread may access a shared memory through variables which are classified as *low* (public), or *high* (private). We want to ensure that concurrent programs cannot copy the contents of high variables to low variables.

Now suppose `x` is a high variable whose value is either `0` or `1`, `y` is a low variable and `c` is some command that takes many steps to complete. Then consider the following program:

- Thread α:
```
if x = 1 then (c;c);
y := 1
```
- Thread β:
```
c;
y := 0
```

The program is well typed in the secure flow system of [15], so it satisfies a possibilistic noninterference prop-

[†]This is a corrected version of the paper that appeared in the Proceedings of the 11th IEEE Computer Security Foundations Workshop, Rockport, MA, 9–11 June 1998, pages 34–43.

[1]This material is based upon activities supported by DARPA and by the National Science Foundation under Agreement Nos. CCR-9612176 and CCR-9612345.

erty. Changing the initial value of x does not change the set of possible final values for y.

But suppose the two threads are scheduled by flipping a coin. Then the threads run at roughly the same rate and the value of x ends up being copied into y with high probability. So there is probabilistic interference when thread scheduling obeys a probability distribution, even when the program is well typed. A change in the initial value of x changes the *probability distribution* of final values of y.

One obvious way to treat the program is through the type system. We might adopt the severe restriction that guards of conditionals be low. In this case, the example is rejected because it is no longer well typed. Another approach is to require that the conditional be executed asynchronously [8]. But there are cases where you want a conditional to execute synchronously.

Another strategy is to extend the language in some way that allows one to use high guards in conditionals, provided a certain (machine-checkable) condition is satisfied. This is the approach we take. In fact, the condition we impose is very simple. We require that conditionals with high guards be executed atomically. This is accomplished by wrapping the conditional with a new command, called **protect** [14], that guarantees the conditional will be executed atomically in a multithreaded environment. We will show that such well-typed programs satisfy a probabilistic noninterference property, which says that the probability distribution of the final values of low variables is independent of the initial values of high variables. In general, the property requires that any total command with a high guard must be protected. These commands include primitive recursion and other forms of guarded statements found in programming languages.

2. Syntax and semantics

Threads are expressed in a simple imperative language:

$$\begin{array}{lrcl}
(\textit{phrases}) & p & ::= & e \mid c \\
(\textit{expressions}) & e & ::= & x \mid n \mid e_1 + e_2 \mid \\
 & & & e_1 - e_2 \mid e_1 = e_2 \\
(\textit{commands}) & c & ::= & x := e \mid c_1 ; c_2 \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & & \textbf{while } e \textbf{ do } c \mid \\
 & & & \textbf{protect } c
\end{array}$$

Metavariable x ranges over identifiers and n over integer literals. Integers are the only values; we use 0 for false and nonzero for true. Note that expressions do not have side effects, nor do they contain partial operations like division.

We define a small-step transition semantics for individual threads in Figure 1. We assume that expressions are evaluated atomically.[2] Thus we simply extend a memory μ in the obvious way to map expressions to integers, writing $\mu(e)$ to denote the value of expression e in memory μ. These rules define a transition relation $\xrightarrow{s}$ on configurations. A *configuration* is either a pair (c, μ) or simply a memory μ. In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ.

At most one thread can be in a protected section at any time. We capture this property by appealing to a standard natural semantics for commands in the hypothesis of rule ATOMICITY, written here as $\mu \vdash c \Rightarrow \mu'$. The hypothesis means that command c evaluates completely to a memory μ' from a memory μ. This is the trick for expressing the atomicity of command execution that allows for a simple noninterference proof. Our natural semantics is standard and is described in [17]. Further, we assume that protected sections are not nested. This is a reasonable assumption since protected sections are transparent in a sequential language, which is what the natural semantics treats. Thus we avoid having to introduce a rule for **protect** into the natural semantics. Finally, we assume that no **while** command occurs in a protected section. The reason for this is to simplify our probabilistic semantics. With **protect**, execution of a thread may block:

$$\textbf{protect while } \textit{true} \textbf{ do } \textit{skip}$$

One needs to compute the probability of a thread being selected from among the unblocked threads only. By prohibiting the potential for nontermination in a protected section, we are guaranteed that all threads in a thread pool are unblocked in that each can make a transition under $\xrightarrow{s}$. Thus, the probability of a thread being selected from a thread pool O can be determined simply from the size of the pool ($|O|$).

As in [15], we take a concurrent program to be a set O of commands that run concurrently. The set O is called the thread pool and it does not grow during execution. We represent O as a mapping from thread identifiers (α, β, ...) to commands. In addition, there is a single global memory μ, shared by all threads, that maps identifiers to integers. Threads communicate via the shared memory. We call a pair (O, μ), a *global configuration*. Execution of a concurrent program takes place under a fixed probability distribution

[2]The noninterference property we prove does not depend on atomicity here unless the time it takes to evaluate an expression depends on the values of high variables.

$$(\textsc{update}) \qquad \frac{x \in \mathit{dom}(\mu)}{(x := e, \mu) \xrightarrow{s} \mu[x := \mu(e)]}$$

$$(\textsc{sequence}) \qquad \frac{(c_1, \mu) \xrightarrow{s} \mu'}{(c_1; c_2, \mu) \xrightarrow{s} (c_2, \mu')}$$

$$\frac{(c_1, \mu) \xrightarrow{s} (c_1', \mu')}{(c_1; c_2, \mu) \xrightarrow{s} (c_1'; c_2, \mu')}$$

$$(\textsc{branch}) \qquad \frac{\mu(e) \text{ nonzero}}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{s} (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \xrightarrow{s} (c_2, \mu)}$$

$$(\textsc{loop}) \qquad \frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{s} \mu}$$

$$\frac{\mu(e) \text{ nonzero}}{(\textbf{while } e \textbf{ do } c, \mu) \xrightarrow{s} (c; \textbf{while } e \textbf{ do } c, \mu)}$$

$$(\textsc{atomicity}) \qquad \frac{\mu \vdash c \Rightarrow \mu'}{(\textbf{protect } c, \mu) \xrightarrow{s} \mu'}$$

$$(\textsc{global}) \qquad \frac{\begin{array}{l} O(\alpha) = c \\ (c, \mu) \xrightarrow{s} \mu' \\ p = 1/|O| \end{array}}{(O, \mu) \xrightarrow{g}_p (O - \alpha, \mu')}$$

$$\frac{\begin{array}{l} O(\alpha) = c \\ (c, \mu) \xrightarrow{s} (c', \mu') \\ p = 1/|O| \end{array}}{(O, \mu) \xrightarrow{g}_p (O[\alpha := c'], \mu')}$$

$$(\{\ \}, \mu) \xrightarrow{g}_1 (\{\ \}, \mu)$$

Figure 1. Sequential and concurrent transition semantics

for the scheduling of threads; our semantics prescribes a uniform distribution for simplicity. The execution is defined by rule GLOBAL, which lets us prove judgments of the form

$$(O, \mu) \xrightarrow{g}_p (O', \mu').$$

This asserts that the probability of going from (O, μ) to (O', μ') is p. The first two GLOBAL rules in Figure 1 specify the global transitions that can be made by a thread pool. The third GLOBAL rule is introduced to accommodate our notion of a probabilistic state. As we shall see, it ensures that probabilities of a state sum to 1. With these GLOBAL rules, we can represent a concurrent program as a discrete Markov chain [3]. The states of the Markov chain are global configurations and the stochastic matrix is determined by $\xrightarrow{g}_p$.

3. The type system

Here are the types used by our type system:

$$\begin{array}{lrcl} (\textit{data types}) & \tau & ::= & L \mid H \\ (\textit{phrase types}) & \rho & ::= & \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd} \end{array}$$

For simplicity, we limit the security classes here to just L and H; it is possible to generalize to an arbitrary partial order of security classes.

The type system is the system of [15], extended with a rule for **protect**. Its rules are given in Figure 2. The rules allow us to prove *typing judgments* of the form $\gamma \vdash p : \rho$ as well as *subtyping judgments* of the form $\rho_1 \subseteq \rho_2$. Here γ denotes an *identifier typing*, which is a finite function from identifiers to phrase types. Note that guards of conditionals may be high.

If $\gamma \vdash c : \rho$ for some ρ, then we say that c is *well typed under* γ. Also, if $O(\alpha)$ is well typed under γ for every $\alpha \in \textit{dom}(O)$, then we say that O is well typed under γ.

4. Probabilistic states

Loosely speaking, our formulation of probabilistic noninterference is a sort of probabilistic lock step execution statement. Under two memories that may differ on high variables, we want to know that the probability that a concurrent program can reach some global configuration under one of the memories is the same as the probability that it reaches an equivalent configuration under the other.

A concurrent program is represented as a discrete Markov chain [3], the states of which are global configurations (O, μ). The stochastic matrix T of the Markov chain is determined by the relation $\xrightarrow{g}_p$. For example, consider the following program:

$$O = \{\alpha := \textbf{while}\ l = 0\ \textbf{do}\ \textit{skip},\ \ \beta := (l := 1)\}$$

The program can get into at most five different configurations, and so its Markov chain has five states, given in Figure 3. The stochastic matrix T for this Markov chain is given in Figure 4. The probability of a transition from state 1, for instance, to state 2 is 1/2 because $p = 1/2$ in the hypothesis of the first GLOBAL rule, the rule that allows this transition to occur.

	1	2	3	4	5
1	0	1/2	1/2	0	0
2	0	0	0	0	1
3	1/2	0	0	1/2	0
4	0	1	0	0	0
5	0	0	0	0	1

Figure 4. Stochastic matrix

The set of Markov states may be countably infinite (a simple example is a nonterminating loop that increments a variable). In this case, the stochastic matrix is also countably infinite. In general, if T is a stochastic matrix and $T((O, \mu), (O', \mu')) > 0$, for some global configurations (O, μ) and (O', μ'), then either O is nonempty and $T((O, \mu), (O', \mu')) = 1/|O|$, or O and O' are empty, $\mu = \mu'$, and $T((O, \mu), (O', \mu')) = 1$.

Kozen uses measures to capture the distributions of values of variables in probabilistic programs [10]. Our strategy is similar. Using the Markov chain, we can model the execution of a concurrent program deterministically as a sequence of *probabilistic states*.

Definition 4.1 *A* probabilistic state *is a probability measure on the set of global configurations.*

A probabilistic state can be represented as a row vector, whose components must sum to 1. So if T is a stochastic matrix and s is a probabilistic state, then the next probabilistic state in the sequence of such states modeling a concurrent computation is simply the vector-matrix product sT. For instance, the initial probabilistic state for the program O in our preceding example, with five states, is $(1\ \ 0\ \ 0\ \ 0\ \ 0)$. It indicates that the Markov chain begins in state 1 with certainty. The next state is given by taking the product of this state with the stochastic matrix of Figure 4, giving $(0\ \ 1/2\ \ 1/2\ \ 0\ \ 0)$. This state indicates the Markov chain can be in states 2 and 3, each with a probability of 1/2. Multiplying this vector by T, we get the third probabilistic state, $(1/4\ \ 0\ \ 0\ \ 1/4\ \ 1/2)$; we can determine the complete execution in this way. The first

(IDENT) $$\frac{\gamma(x) = \rho}{\gamma \vdash x : \rho}$$

(INT) $$\gamma \vdash n : \tau$$

(R-VAL) $$\frac{\gamma \vdash e : \tau\ var}{\gamma \vdash e : \tau}$$

(SUM) $$\frac{\gamma \vdash e_1 : \tau,\ \ \gamma \vdash e_2 : \tau}{\gamma \vdash e_1 + e_2 : \tau}$$

(ASSIGN) $$\frac{\gamma \vdash x : \tau\ var,\ \ \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ cmd}$$

(COMPOSE) $$\frac{\gamma \vdash c_1 : \tau\ cmd,\ \ \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash c_1 ; c_2 : \tau\ cmd}$$

(IF) $$\frac{\gamma \vdash e : \tau,\ \ \gamma \vdash c_1 : \tau\ cmd,\ \ \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ cmd}$$

(WHILE) $$\frac{\gamma \vdash e : L,\ \ \gamma \vdash c : L\ cmd}{\gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : L\ cmd}$$

(PROTECT) $$\frac{\gamma \vdash c : \tau\ cmd}{\gamma \vdash \mathbf{protect}\ c : \tau\ cmd}$$

(BASE) $$L \subseteq H$$

(REFLEX) $$\rho \subseteq \rho$$

(CMD$^-$) $$\frac{\tau_1 \subseteq \tau_2}{\tau_2\ cmd \subseteq \tau_1\ cmd}$$

(SUBTYPE) $$\frac{\gamma \vdash p : \rho_1,\ \ \rho_1 \subseteq \rho_2}{\gamma \vdash p : \rho_2}$$

Figure 2. Typing and subtyping rules

$$\begin{array}{lll}
1) & (\{\alpha := \mathbf{while}\ l = 0\ \mathbf{do}\ skip,\ \ \beta := (l := 1)\}, & [l := 0]) \\
2) & (\{\alpha := \mathbf{while}\ l = 0\ \mathbf{do}\ skip\}, & [l := 1]) \\
3) & (\{\alpha := skip; \mathbf{while}\ l = 0\ \mathbf{do}\ skip,\ \ \beta := (l := 1)\}, & [l := 0]) \\
4) & (\{\alpha := skip; \mathbf{while}\ l = 0\ \mathbf{do}\ skip\}, & [l := 1]) \\
5) & (\{\ \}, & [l := 1])
\end{array}$$

Figure 3. States of Markov chain

$$
\begin{array}{c}
\{(\{\alpha := \textbf{while } l = 0 \textbf{ do } \mathit{skip},\ \ \beta := (l := 1)\}, [l := 0]) \,:\, 1\} \\
\downarrow \\
\left\{ \begin{array}{l}
(\{\alpha := \textbf{while } l = 0 \textbf{ do } \mathit{skip}\}, [l := 1]) \,:\, 1/2, \\
(\{\alpha := \mathit{skip}; \textbf{while } l = 0 \textbf{ do } \mathit{skip},\ \ \beta := (l := 1)\}, [l := 0]) \,:\, 1/2
\end{array} \right\} \\
\downarrow \\
\left\{ \begin{array}{l}
(\{\,\}, [l := 1]) \,:\, 1/2, \\
(\{\alpha := \textbf{while } l = 0 \textbf{ do } \mathit{skip},\ \ \beta := (l := 1)\}, [l := 0]) \,:\, 1/4, \\
(\{\alpha := \mathit{skip}; \textbf{while } l = 0 \textbf{ do } \mathit{skip}\}, [l := 1]) \,:\, 1/4
\end{array} \right\} \\
\downarrow \\
\left\{ \begin{array}{l}
(\{\,\}, [l := 1]) \,:\, 1/2, \\
(\{\alpha := \mathit{skip}; \textbf{while } l = 0 \textbf{ do } \mathit{skip},\ \ \beta := (l := 1)\}, [l := 0]) \,:\, 1/8, \\
(\{\alpha := \textbf{while } l = 0 \textbf{ do } \mathit{skip}\}, [l := 1]) \,:\, 3/8
\end{array} \right\} \\
\downarrow \\
\left\{ \begin{array}{l}
(\{\,\}, [l := 1]) \,:\, 7/8, \\
(\{\alpha := \textbf{while } l = 0 \textbf{ do } \mathit{skip},\ \ \beta := (l := 1)\}, [l := 0]) \,:\, 1/16, \\
(\{\alpha := \mathit{skip}; \textbf{while } l = 0 \textbf{ do } \mathit{skip}\}, [l := 1]) \,:\, 1/16
\end{array} \right\}
\end{array}
$$

Figure 5. A probabilistic state sequence

five probabilistic states in the sequence are depicted in Figure 5. The fifth state, for instance, tells us that the probability that O terminates under memory $[l := 0]$ in at most four steps is 7/8.

Thread pool O is an example of a concurrent program that is probabilistically total since it halts with probability 1, but is not nondeterministically total for it has an infinite computation path.

Note that although there may be infinitely many states in the Markov chains corresponding to our programs, the probabilistic states that arise in our program executions will only assign nonzero probability to finitely many of them. This is because we begin execution in a single global configuration (O, μ), and we only branch by at most a factor of k at each step, where k is the number of threads in O. If we were to extend our language with a random number generator that returns an arbitrary integer with respect to some probability distribution, then we would have to consider probabilistic states which give nonzero probabilities to an infinite number of global configurations.

With probabilistic states, we can now see how probability distributions can be sensitive to initial values of high variables, even for programs that have types in the system of Figure 2. Consider the example in the introduction where c is instantiated to *skip*:

$$O = \left\{ \begin{array}{l} \alpha := (\textbf{if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1, \\ \beta := (\mathit{skip};\ y := 0) \end{array} \right\}$$

Each thread is well typed, assuming *skip* has type *H cmd*. We give two sequences of state transitions. One begins with x equal to 0 (Figure 6) and the other with x equal to 1 (Figure 7). Notice the change in distribution for the final values of y when the initial value of the high variable x changes. For instance, the probability that y has final value 1 when x equals 1 is 13/16, and falls to 1/2 when x equals 0. What is going on here is that the initial value of x affects the amount of time required to execute the conditional; this in turn affects the likely order in which the two assignments to y are executed. Now suppose that we protect the conditional in this example. Then the conditional (in effect) executes in one step, regardless of the value of x, and so the sequence of transitions for $x = 0$ is equivalent, state by state, to the sequence of transitions for $x = 1$ (Figures 8 and 9).

5. Probabilistic noninterference

Now we are ready to prove our main result. We begin with two lemmas which are proved in [17]:

Lemma 5.1 (Simple Security) *If $\gamma \vdash e : L$, then $\gamma(x) = L$ for every identifier x in e.*

Lemma 5.2 (Confinement) *If $\gamma \vdash c : H\ cmd$, then $\gamma(x) = H\ var$ for every identifier x assigned to in c.*

Definition 5.1 (Protected) *A command is* protected *if every conditional in the command with a guard of type H falls within the scope of a* **protect**.

Definition 5.2 *Given an identifier typing γ, we say that memories μ and ν are equivalent, written $\mu \sim_\gamma \nu$, if μ, ν, and γ have the same domain and μ and ν agree on all L identifiers.*

$$\{(\{\alpha := (\textbf{if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 0, y := 0])\ :\ 1\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := (\textbf{if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := y := 0\}, [x := 0, y := 0])\ :\ 1/2, \\ (\{\alpha := y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 0, y := 0])\ :\ 1/2 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := (\textbf{if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1\}, [x := 0, y := 0])\ :\ 1/4, \\ (\{\alpha := y := 1,\ \ \beta := y := 0\}, [x := 0, y := 0])\ :\ 1/2, \\ (\{\beta := (\mathit{skip};\ y := 0)\}, [x := 0, y := 1])\ :\ 1/4 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := y := 1\}, [x := 0, y := 0])\ :\ 1/2, \\ (\{\beta := y := 0\}, [x := 0, y := 1])\ :\ 1/2 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\ \}, [x := 0, y := 1])\ :\ 1/2, \\ (\{\ \}, [x := 0, y := 0])\ :\ 1/2 \end{array}\right\}$$

Figure 6. Probabilistic state sequence when $x = 0$

$$\{(\{\alpha := (\textbf{if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 0])\ :\ 1\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := (\textbf{if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := y := 0\}, [x := 1, y := 0])\ :\ 1/2, \\ (\{\alpha := (\mathit{skip}; \mathit{skip}); y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 0])\ :\ 1/2 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := (\textbf{if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1\}, [x := 1, y := 0])\ :\ 1/4, \\ (\{\alpha := (\mathit{skip}; \mathit{skip}); y := 1,\ \ \beta := y := 0\}, [x := 1, y := 0])\ :\ 1/2, \\ (\{\alpha := \mathit{skip}; y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 0])\ :\ 1/4 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := (\mathit{skip}; \mathit{skip}); y := 1\}, [x := 1, y := 0])\ :\ 1/2, \\ (\{\alpha := \mathit{skip}; y := 1,\ \ \beta := y := 0\}, [x := 1, y := 0])\ :\ 3/8, \\ (\{\alpha := y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 0])\ :\ 1/8 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := \mathit{skip}; y := 1\}, [x := 1, y := 0])\ :\ 11/16, \\ (\{\alpha := y := 1,\ \ \beta := y := 0\}, [x := 1, y := 0])\ :\ 1/4, \\ (\{\beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 1])\ :\ 1/16 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\alpha := y := 1\}, [x := 1, y := 0])\ :\ 13/16, \\ (\{\beta := y := 0\}, [x := 1, y := 1])\ :\ 3/16 \end{array}\right\}$$
$$\downarrow$$
$$\left\{\begin{array}{l} (\{\ \}, [x := 1, y := 1])\ :\ 13/16, \\ (\{\ \}, [x := 1, y := 0])\ :\ 3/16 \end{array}\right\}$$

Figure 7. Probabilistic state sequence when $x = 1$

$$\begin{gathered}
\{(\{\alpha := (\textbf{protect if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 0, y := 0])\ :\ 1\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\alpha := (\textbf{protect if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := y := 0\}, [x := 0, y := 0])\ :\ 1/2, \\
(\{\alpha := y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 0, y := 0])\ :\ 1/2
\end{array}\right\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\alpha := (\textbf{protect if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1\}, [x := 0, y := 0])\ :\ 1/4, \\
(\{\alpha := y := 1,\ \ \beta := y := 0\}, [x := 0, y := 0])\ :\ 1/2, \\
(\{\beta := (\mathit{skip};\ y := 0)\}, [x := 0, y := 1])\ :\ 1/4
\end{array}\right\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\alpha := y := 1\}, [x := 0, y := 0])\ :\ 1/2, \\
(\{\beta := y := 0\}, [x := 0, y := 1])\ :\ 1/2
\end{array}\right\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\ \}, [x := 0, y := 1])\ :\ 1/2, \\
(\{\ \}, [x := 0, y := 0])\ :\ 1/2
\end{array}\right\}
\end{gathered}$$

Figure 8. Probabilistic state sequence when $x = 0$

$$\begin{gathered}
\{(\{\alpha := (\textbf{protect if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 0])\ :\ 1\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\alpha := (\textbf{protect if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1,\ \ \beta := y := 0\}, [x := 1, y := 0])\ :\ 1/2, \\
(\{\alpha := y := 1,\ \ \beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 0])\ :\ 1/2
\end{array}\right\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\alpha := (\textbf{protect if } x = 1 \textbf{ then } \mathit{skip}; \mathit{skip});\ y := 1\}, [x := 1, y := 0])\ :\ 1/4, \\
(\{\alpha := y := 1,\ \ \beta := y := 0\}, [x := 1, y := 0])\ :\ 1/2, \\
(\{\beta := (\mathit{skip};\ y := 0)\}, [x := 1, y := 1])\ :\ 1/4
\end{array}\right\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\alpha := y := 1\}, [x := 1, y := 0])\ :\ 1/2, \\
(\{\beta := y := 0\}, [x := 1, y := 1])\ :\ 1/2
\end{array}\right\} \\
\downarrow \\
\left\{\begin{array}{l}
(\{\ \}, [x := 1, y := 1])\ :\ 1/2, \\
(\{\ \}, [x := 1, y := 0])\ :\ 1/2
\end{array}\right\}
\end{gathered}$$

Figure 9. Probabilistic state sequence when $x = 1$

We now show that if we execute a well-typed, protected command c in two equivalent memories, the two executions proceed in lock step:

Lemma 5.3 (Lock Step Execution) *Suppose c is well typed under γ and protected, and that $\mu \sim_\gamma \nu$. If $(c, \mu) \xrightarrow{s} (c', \mu')$, then there exists ν' such that $(c, \nu) \xrightarrow{s} (c', \nu')$ and $\mu' \sim_\gamma \nu'$. And if $(c, \mu) \xrightarrow{s} \mu'$, then there exists ν' such that $(c, \nu) \xrightarrow{s} \nu'$ and $\mu' \sim_\gamma \nu'$.*

Proof. By induction on the structure of c. The interesting cases are the **protect** command and conditionals. In particular, we need only consider conditionals with guards of type L since those with guards of type H are protected and therefore fall under the **protect** case.

For conditionals with guards e of type L, the theorem follows from Lemma 5.1 which guarantees that $\mu(e) = \nu(e)$, and therefore evaluation of the conditional under ν may proceed along the same branch as the evaluation under μ.

Now suppose $(\textbf{protect}\ c, \mu) \xrightarrow{s} \mu'$ and $\mu \sim_\gamma \nu$. Then by rule ATOMICITY,

$$\mu \vdash c \Rightarrow \mu'$$

By the Termination Agreement theorem (Theorem 3.1, [16]), there is a memory ν' such that $\nu \vdash c \Rightarrow \nu'$ and $\mu' \sim_\gamma \nu'$. Thus, $(\textbf{protect}\ c, \nu) \xrightarrow{s} \nu'$. □

Now we wish to extend the Lock Step Execution lemma to probabilistic states. First, we need a notion of equivalence among probabilistic states. The basic idea is that two probabilistic states are equivalent under γ if they are the same after any high variables are projected out. Suppose, for example, that $x : H$ and $y : L$. Then

$$\left\{ \begin{array}{l} (O, [x := 0, y := 0]) : 1/3, \\ (O, [x := 1, y := 0]) : 1/3, \\ (O', [x := 0, y := 1]) : 1/3 \end{array} \right\}$$

is equivalent to

$$\{(O, [x := 2, y := 0]) : 2/3, (O', [x := 3, y := 1]) : 1/3\},$$

because in each case the result of projecting out the high variable x is

$$\{(O, [y := 0]) : 2/3, (O', [y := 1]) : 1/3\}.$$

Notice that projecting out high variables can cause several configurations to collapse into one, requiring that their probabilities be summed. More formally, we define equivalence as follows:[3]

[3]Definition 5.3 here differs from the one in the workshop proceedings. The one in the proceedings is incorrect.

Definition 5.3 *Given identifier typing γ and memory μ, let μ_γ denote the result of erasing all high variables from μ. And given probabilistic state s, let the projection of s onto the low variables of γ, denoted s_γ, be defined by*

$$s_\gamma(O, \mu_\gamma) = \sum_{\nu \text{ such that } \nu \sim_\gamma \mu} s(O, \nu)$$

Finally, we say that probabilistic states s and s' are equivalent under γ, written $s \sim_\gamma s'$, if $s_\gamma = s'_\gamma$.

Next we say that a probabilistic state s is well typed and protected under γ if for every global configuration (O, μ) with $s(O, \mu) > 0$, every thread in O is well typed and protected under γ, and $dom(\mu) = dom(\gamma)$.

For any global configuration (O, μ), the *point mass on* (O, μ), denoted $\iota_{(O,\mu)}$, is the probabilistic state that that gives probability 1 to (O, μ) and probability 0 to all other global configurations.

Now we can show, as a corollary to the Lock Step Execution lemma, that $\sim_\gamma$ is a congruence with respect to the stochastic matrix T on well-typed, protected point masses.

Lemma 5.4 (Congruence on Point Masses) *If ι and ι' are well-typed, protected point masses such that $\iota \sim_\gamma \iota'$, then $\iota T \sim_\gamma \iota' T$.*

Proof. Since $\iota \sim_\gamma \iota'$, there must exist a thread pool O and memories μ and ν such that $\iota = \iota_{(O,\mu)}$, $\iota' = \iota_{(O,\nu)}$, and $\mu \sim_\gamma \nu$.

If $O = \{\ \}$, then by third (GLOBAL) rule, we see that $\iota T = \iota$ and $\iota' T = \iota'$. So $\iota T \sim_\gamma \iota' T$.

Now suppose that O is nonempty. We show that for every (O', μ') where $(\iota T)(O', \mu') > 0$, there is a ν' such that $\mu' \sim_\gamma \nu'$ and $(\iota T)(O', \mu') = (\iota' T)(O', \nu')$. So suppose (O', μ') is a global configuration and $(\iota T)(O', \mu') > 0$. Since ι is a point mass,

$$(\iota T)(O', \mu') = T((O, \mu), (O', \mu'))$$

Therefore, $T((O, \mu), (O', \mu')) > 0$. By the definition of T, then, $T((O, \mu), (O', \mu')) = 1/|O|$ and there is a thread α and command c such that $O(\alpha) = c$ and either

1. $(c, \mu) \xrightarrow{s} (c', \mu')$ and $O' = O[\alpha := c']$, or else
2. $(c, \mu) \xrightarrow{s} \mu'$ and $O' = O - \alpha$.

In the first case, we have, by the Lock Step Execution lemma, that there exists ν' such that $(c, \nu) \xrightarrow{s} (c', \nu')$ and $\mu' \sim_\gamma \nu'$. Then, by rule (GLOBAL), $(O, \nu) \xrightarrow{g}_{1/|O|} (O[\alpha := c'], \nu')$, so by definition of T,

$$T((O, \nu), (O', \nu')) = 1/|O|$$

But ι' is also a point mass, therefore

$$(\iota' T)(O', \nu') = T((O, \nu), (O', \nu'))$$

Thus, $(\iota T)(O', \mu') = 1/|O| = (\iota' T)(O', \nu')$. The second case above is similar.

So for a given configuration (O, μ), if $\mu \sim_\gamma \nu$ and $(\iota T)(O, \nu) > 0$, then there exists ν' such that $\nu' \sim_\gamma \nu$ and $(\iota' T)(O, \nu') = (\iota T)(O, \nu)$ from above. Since $\nu' \sim_\gamma \mu$, $(\iota' T)(O, \nu')$ must be in the sum $(\iota' T)_\gamma(O, \mu_\gamma)$. Therefore, $(\iota T)_\gamma(O, \mu_\gamma) \leq (\iota' T)_\gamma(O, \mu_\gamma)$. Symmetrically, we have $(\iota T)_\gamma(O, \mu_\gamma) \geq (\iota' T)_\gamma(O, \mu_\gamma)$ and so $(\iota T)_\gamma = (\iota' T)_\gamma$, or $\iota T \sim_\gamma \iota' T$. $\square$

Now we wish to generalize the above Congruence lemma from point masses to arbitrary probabilistic states; this generalization is a direct consequence of the linearity of T. More precisely, the set of all measures forms a vector space if we define

- $(s + s')(O, \mu) = s(O, \mu) + s'(O, \mu)$, for measures s and s', and
- $(as)(O, \mu) = a(s(O, \mu))$, for real a and measure s.

With respect to this vector space, T is a linear transformation. Furthermore, $\sim_\gamma$ respects the vector space operations:

Lemma 5.5 *If $s_i \sim_\gamma s_i'$ for all i, then*

$$a_1 s_1 + a_2 s_2 + a_3 s_3 + \cdots \quad \sim_\gamma \quad a_1 s_1' + a_2 s_2' + a_3 s_3' + \cdots$$

Theorem 5.6 (Probabilistic Noninterference) *If s and s' are well-typed, protected probabilistic states such that $s \sim_\gamma s'$, then $sT \sim_\gamma s'T$.*

Proof. To begin with, we argue that s and s' can be expressed as (possibly countably infinite) linear combinations of (not necessarily distinct) point masses such that the corresponding coefficients are the same, and the corresponding point masses are equivalent.

Now, we know that we can express s and s' as linear combinations of point masses:

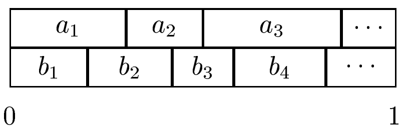
$$s = a_1 \iota_1 + a_2 \iota_2 + a_3 \iota_3 + \cdots$$

and

$$s' = b_1 \iota_1' + b_2 \iota_2' + b_3 \iota_3' + \cdots$$

Assume, for now, that s_γ (and s_γ') is a point mass. That is, $\iota_i \sim_\gamma \iota_j \sim_\gamma \iota_i' \sim_\gamma \iota_j'$ for all i and j.

Note that the a_i's and b_i's both sum to 1; hence they both can be understood as partitioning the unit interval $[0, 1]$:



To unify the coefficients in the two linear combinations, we must take the *union* of the two partitions, splitting up any terms that cross partition boundaries. For example, based on the picture above we would split the term $a_1 \iota_1$ of s into $b_1 \iota_1 + (a_1 - b_1)\iota_1$. And we would split the term $b_2 \iota_2'$ of s' into $(a_1 - b_1)\iota_2' + (b_2 - (a_1 - b_1))\iota_2'$. Continuing in this way, we can unify the coefficients of s and s'.

We can describe the splitting process more precisely as follows. We simultaneously traverse s and s', splitting terms as we go. Let $a\iota$ and $b\iota'$ be the next terms to be unified. If $a = b$, then keep both these terms unchanged. If $a < b$, then keep term $a\iota$ in s, but split $b\iota'$ into $a\iota'$ and $(b - a)\iota'$ in s'. Handle the case $a > b$ symmetrically. If one or both of the sums are infinite, then of course the algorithm gives an infinite sum. But each term of s and of s' is split only finitely often (otherwise the a_i's and b_i's would not have the same sum) with one exception—if s is a finite sum and s' is an infinite sum, then the last term of s is split into an infinite sum.

So far, we have shown how to unify the coefficients of s and s' in the case where s_γ (and s_γ') is a point mass. In the general case, s and s' must first be rearranged into sums of sums of equivalent point masses:

$$s = (a_{11}\iota_{11} + a_{12}\iota_{12} + \cdots) + (a_{21}\iota_{21} + a_{22}\iota_{22} + \cdots) + \cdots$$

and

$$s' = (b_{11}\iota_{11}' + b_{12}\iota_{12}' + \cdots) + (b_{21}\iota_{21}' + b_{22}\iota_{22}' + \cdots) + \cdots$$

where $\iota_{ij} \sim_\gamma \iota_{ik} \sim_\gamma \iota_{ij}' \sim_\gamma \iota_{ik}'$ for all i, j, and k. Also, for each i, $\sum_j a_{ij} = \sum_j b_{ij}$. Hence we can apply the algorithm above to unify the a_{1j}'s with the b_{1j}'s, the a_{2j}'s with the b_{2j}'s, and so forth. Then we can form a single sum for s and for s' by interleaving these sums in a standard way.

The final result of all this effort is that we can express s and s' as

$$s = c_1 \iota_1'' + c_2 \iota_2'' + c_3 \iota_3'' + \cdots$$

and

$$s' = c_1 \iota_1''' + c_2 \iota_2''' + c_3 \iota_3''' + \cdots$$

where $\iota_i'' \sim_\gamma \iota_i'''$ for all i. Now, since T is a linear transformation, we have

$$sT = c_1(\iota_1'' T) + c_2(\iota_2'' T) + c_3(\iota_3'' T) + \cdots$$

and

$$s'T = c_1(\iota_1'''T) + c_2(\iota_2'''T) + c_3(\iota_3'''T) + \cdots$$

By the Congruence on Point Masses Lemma, we have $\iota_i''T \sim_\gamma \iota_i'''T$, for all i. So, by the lemma above, $sT \sim_\gamma s'T$. $\square$

6. Discussion

The need for a probabilistic view of security in nondeterministic computer systems has been understood for some time [18, 12]. Security properties (models) to treat probabilistic channels in nondeterministic systems have been formulated by McLean[11] and Gray [6, 7]. It is important, however, to recognize that these efforts address a different problem than what we consider in this paper. They consider a computer system with a number of *users*, classified as high or low, who send inputs to and receive outputs from the system. The problem is to prevent high users, who have access to high information, from communicating with low users, who should have access only to low information. What makes treating privacy in this setting especially difficult is that users need not be processes under control of the system—they may be *people*, who are *external* to the system and who can observe the system's behavior from the outside. As a result, a high user may be able to communicate covertly by modulating system performance to encode high information that a low user in turn decodes using a real-time clock outside the system. Furthermore, because the low user is measuring *real time*, the modulations can depend on low-level system implementation details, such as the paging and caching behavior of the underlying hardware. This implies that it is not enough to prove privacy with respect to a high-level, abstract system semantics (like the semantics of Figure 1). To guarantee privacy, it is necessary for the system model to address all the implementation details.

In a mobile-code framework, where hosts are trusted, ensuring privacy is more tractable. A key assumption here is that any attempt to compromise privacy must arise from within the mobile code, which is *internal* to the system. As a result, the system can control what the mobile code can do and what it can observe. For example, if mobile-code threads are not allowed to see a real-time clock, then they can measure the timing of other threads only by observing variations in thread interleavings. Hence, assuming a correct implementation of the semantics in Figure 1, threads will not be able to detect any variations in the running time of a protected command, nor will they be able to detect timing variations due to low-level implementation details. Consequently, timing attacks are impossible in well-typed, protected programs in our language. For instance, Kocher describes a timing attack on RSA [9]. Basically, he argues that an attacker can discover a private key x by observing the amount of time required by several modular exponentiations $y^x \bmod n$. Under our framework, the modular exponentiation would be protected,[4] which means that no useful timing information about exponentiation would be available to other threads—it would always appear to execute in exactly one step.

7. Other related research

Other work in secure information flow, in a parallel setting, includes that of Andrews and Reitman [1], Melliar-Smith and Moser [13], Focardi and Gorrieri [4, 5], and Banatre and Bryce [2]. Melliar-Smith and Moser consider covert channels in Ada. They describe a data dependency analysis to find places where Ada programs depend on the relative timing of operations within a system. Andrews and Reitman give an axiomatic flow logic for treating information flow in the presence of process synchronization. They also sketch how one might treat timing channels in the logic. Banatre and Bryce give an axiomatic flow logic for CSP processes, also treating information flow arising from synchronization. None of these efforts, though, gives a satisfactory account of the security properties that they guarantee. Focardi and Gorrieri identify trace-based and bisimulation-based security properties for systems expressed in an extension of Milner's CCS, which they call the Security Process Algebra. These properties, however, are possibilistic in nature (e.g. a system is *SNNI* [5] if the set of traces that a low observer can see of a system is possible regardless of whether high-level actions are enabled or disabled in the system).

8. Conclusion

So what is the significance of our result? It depends on what can be observed. With respect to internal program behavior, our Probabilistic Noninterference result rules out all covert flows from high variables to low variables. But if *external* observation of the running program is allowed, then of course covert channels of the kind discussed in Section 6 remain possible. In this case, more elaborate security properties, like Gray's *information flow security* [7], may be needed. Note,

[4] Because we do not allow **while** commands to occur within protected sections, this requires that we program the modular exponentiation in terms of a primitive recursive looping construct.

however, that the mobile code setting affords us more control over external observations than would normally be possible. When we execute some mobile code on our machine, we can limit communication with the outside world, preventing precise observations of a program's execution time, for example.

References

[1] G. Andrews and R. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[2] J. Banâtre and C. Bryce. Information flow control in a parallel language framework. In *Proceedings 6th IEEE Computer Security Foundations Workshop*, pages 39–52, June 1993.

[3] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley & Sons, Inc., third edition, 1968.

[4] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1994/1995.

[5] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997.

[6] J. W. Gray, III. Probabilistic interference. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, CA, May 1990.

[7] J. W. Gray, III. Toward a mathematical foundation for information flow security. In *Proceedings 1991 IEEE Symposium on Security and Privacy*, pages 21–34, Oakland, CA, May 1991.

[8] N. Heintze and J. Riecke. The SLam Calculus: Programming with secrecy and integrity. In *Proceedings 25th Symposium on Principles of Programming Languages*, pages 365–377, San Diego, CA, Jan. 1998.

[9] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Proceedings 16th Annual Crypto Conference*, Aug. 1996.

[10] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

[11] J. McLean. Security models and information flow. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, 1990.

[12] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.

[13] P. Melliar-Smith and L. Moser. Protection against covert storage and timing channels. In *Proceedings 4th IEEE Computer Security Foundations Workshop*, pages 209–214, June 1991.

[14] H. R. Nielson and F. Nielson. *Semantics with Applications, A Formal Introduction*. Wiley, 1992.

[15] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings 25th Symposium on Principles of Programming Languages*, pages 355–364, San Diego, CA, Jan. 1998.

[16] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[17] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[18] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 144–161, Oakland, CA, May 1990.